Copyright © 1997 IEEE. See full copyright notice at Table of Contents.

An Interface Specification Language for Automatically Analyzing Cryptographic Protocols

Stephen H. Brackin*
Arca Systems, Inc.
ESC/AXS
Hanscom AFB, MA 01731-2116

Abstract

This paper describes a simple Interface Specification Language (ISL) for cryptographic protocols and their desired properties, and an Automatic Authentication Protocol Analyzer (AAPA) that automatically either proves — using an extension of the Gong, Needham, Yahalom belief logic — that specified protocols have their desired properties, or identifies precisely where these proof attempts fail. ISL and the AAPA make it easy for protocol designers to incorporate formal analysis into the protocol design process, where they clarify designs and reveal a large class of common errors. ISL and the AAPA have already shown potential deficiencies in published protocols and been useful in designing new protocols.

1 Introduction

Cryptographic protocols are short sequences of message exchanges, usually involving encryption, intended to establish secure communication over insecure networks. The basic security issues for protocols are *authentication* (i.e., knowing who sent information), and *nondisclosure* (i.e., not revealing information to those not meant to receive it).

Different protocols provide different services, operate under different conditions, and are subject to different speed and memory constraints. Designing a new network service that requires security — e.g., any service involving money — often includes designing a new protocol. Designing a new network service might also involve chosing a protocol that is not new, but new to the network service's designers.

In the worst case of an insecure network, all communication is controlled by an enemy. Any message received can be from the enemy, and any message sent can go to the enemy. Even in this worst case, though, correctly designed and implemented protocols, using secure encryption functions, can guarantee that communication is secure whenever it occurs.

Unfortunately, deciding whether cryptographic protocols actually provide the security they are designed to provide is notoriously difficult. Even if one ignores implementation bugs and attacks based on fast, successful code-breaking, security can still be compromised by clever attacks involving things such as stopping messages, replaying old messages, and modifying messages with information copied or computed from other messages. This is called *protocol failure* [23].

Even though protocol failure has been known and studied for some time, there are still many instances of protocols, including published protocols recommended by experts, that are vulnerable to attack [8, 15]. Gavin Lowe, one of the leading experts on protocol failure, has estimated that over half of published protocols actually fail to guarantee their security goals [14].

Part of the problem is that protocol designers often lack expertise on protocol failure and techniques for avoiding it, e.g., [1]. A deeper problem, though, is that even experts often miss possible attacks.

This paper describes a tool, the Automatic Authentication Protocol Analyzer (AAPA), that partially solves the problem of protocol failure for designers. It inputs formal specifications of protocols and their intended security properties in a simple Interface Specification Language (ISL), and *automatically, in minutes,* either proves that these protocols have their intended properties or identifies precisely where these proof attempts fail.

*The author wishes to thank Grace Hammonds, Randy Lichota, and Jon Millen for their assistance. This work was supported by Rome Laboratory and Air Force Materiel Command's Electronic Systems Center/Software Center (ESC/AXS), Hanscom AFB, through the Portable, Reusable, Integrated Software Modules (PRISM) program, contracts F19628-92-C-0006 and F19628-92-C-0008.

Designers can easily use ISL to specify protocols and their desired properties, then quickly obtain AAPA analyses of these specifications. They can then make indicated changes needed for their protocols, and iterate the analyze/modify process until their protocols' specifications have all their stated properties.

The AAPA produces its proofs using an extension of the Gong, Needham, Yahalom (GNY) protocol analysis belief logic [10, 9], implemented in the Higher Order Logic (HOL) [11] family of proof tools. See [4, 3] for a description of the AAPA's belief logic, and see [3] for a description of its proof-construction algorithm.

The AAPA does not solve all of the protocol-failure problem. It misses some failures, most notably nondisclosure failures; belief logics make authentication deductions by assuming that there have not been nondisclosure failures. The AAPA also does not find explicit attacks exploiting the proof failures that it finds. Section 6 describes the AAPA's strengths and weaknesses with respect to other protocol analysis tools, and Section 7 describes ongoing work to lessen or eliminate the AAPA's weaknesses.

What the AAPA does do, though, is catch broad classes of common errors, doing so quickly and easily enough to make performing an AAPA analysis a worthwhile part of any design effort for a network service requiring authentication. An AAPA analysis can catch potential failures, unreasonable initial-conditions assumptions, and critical issues that should be addressed in protocol documentation [2]. Even if it does not catch a potential failure in a protocol, an AAPA analysis can identify the trust relationships necessary to make the protocol work, identify which parts of the protocol actually convey critical information, and identify redundancies [13]. An ISL specification also serves as a precise, but simple, high-level description of a protocol that clarifies the protocol's documentation and improves communication among members of the protocol's development team.

The AAPA can be used either alone or as part of the `Convince` system, which consists of the AAPA together with an StP-based GUI that automatically creates ISL specifications from user-created graphical protocol representations [5, 12]. `Convince` and the AAPA were used in the development of the Public-Key Kerberos protocol [24, 7, 31].

The remainder of this paper is organized as follows: Section 2 gives basic concepts and terminology for cryptographic protocols in general and the network model assumed by the AAPA in particular. Section 3 gives a sample ISL specification of a recently published, flawed protocol. Section 4 defines the language ISL, its BNF syntax, its concepts and their interpretations, and the semantic restrictions the AAPA imposes on ISL specifications. Section 5 briefly describes the results produced by the AAPA, and illustrates these results with the ISL specification from Section 3. Section 6 compares the AAPA's strengths and weaknesses to those of other protocol analysis tools. Section 7 describes ongoing work to reduce or eliminate the AAPA's weaknesses, and Section 8 gives conclusions.

2 Cryptographic Protocol Basics

This section gives assumptions and terminology for cryptographic protocols in general, and the AAPA's model of cryptographic protocols in particular. It also describes basic techniques used in these protocols.

A *network*, or distributed environment, consists of *principals* connected by communication links. Messages on these links constitute the only communications between principals. In the AAPA's model, any principal can place a message on any link and can see or modify any message on any link.

A *protocol* is a distributed algorithm, carried out by principals acting as state machines. The protocol determines which messages the principals send as functions of their internal states. A protocol is divided into *stages* by message transmissions; the number of stages is always finite. A run, or *session*, is a particular execution of a protocol. The AAPA only considers sessions that seem to end successfully, not considering possible alarm, adaptation, or retry provisions for other sessions.

At each stage of a protocol session, each principal has a finite set of *received* data items, which the principal either had before the session began or extracted from messages sent to it during the session. Each principal also has a potentially infinite set of *possessions*, which are pieces of information that the principal is capable of computing from the data items it has received.

At each stage of a protocol session in the AAPA model, each principal also has a set of *beliefs*. Principals believe a proposition if they can be confident, even though this confidence is not absolute, that this proposition is true. Belief can be based, for instance, on the near impossibility of quickly decrypting encrypted information without having the needed key. The AAPA interprets "believes" as "believes with confidence".

Every principal starts a session with initial sets of received items and beliefs, and expands these sets by receiving messages. The AAPA treats a protocol's stage 0 as giving the protocol's initial conditions, and subsequent stages as giving its message exchanges. In the AAPA's model, every principal's received-items and belief sets increase monotonically during a session, but a principal need not still have a data item or belief that this principal had in an earlier session.

The AAPA assumes that the principals involved in a session do not send or receive messages other than the ones in the session; in particular, it assumes that the principals do not simply give away their secrets.

Cryptographic protocols typically use encryption in one or both of two forms: *symmetric-key* encryption, in which data encrypted with a key is decrypted with the same key; and *public-key* encryption, in which data encrypted with a key is decrypted with a different, mathematically related key. In public-key encryption, one of the two keys in a pair, the *public* key, is made widely available, while the corresponding *private* key is kept secret. It is computationally infeasible to compute a private key from the corresponding public key. See [29] for more information.

Cryptographic protocols also often use time stamps, *nonces* (newly created random values highly unlikely to have been created before), *hash functions* (functions computing seemingly random bit strings as functions of their inputs, with the property that it is computationally infeasible to create an input with a given hash or find two inputs with the same hash), *message authentication codes* (values produced by key-dependent hash functions), and *key-exchange functions* (functions allowing two users to compute a key only they share by combining their own private keys and each others' public keys). See [29] for more information.

3 Sample Protocol Specification

The following ISL specification describes the Tatebayashi, Matsuzakai, Newman protocol [34].

In this protocol, principals `A` and `B` communicate through a trusted server `S`, using public-key encryption, to exchange a newly created symmetric key for subsequent communication between `A` and `B`. `A` begins the protocol by sending `S` its name, the name `B` of the principal it wants to communicate with, and a newly created symmetric key `KsA`, encrypted with `S`'s public key, that `S` is to use for encrypting the symmetric key it obtains from `B`. `S` tells `B` that `A` wants to communicate with it. `B` sends `S` a newly created symmetric key `KsB`, encrypted with `S`'s public key, for the desired communication between `A` and `B`. `S` then completes the protocol by sending `A` the key `KsB`, encrypted using symmetric-key encryption and the key `KsA`.

The protocol's initial conditions assert the following:

- `A`, `B`, and `S` can perform all the various symmetric- and public-key encryption and decryption operations (`Se`, `Sd`, `Pe`, and `Pd`) used. The specification does not depend on the particular operations used, so it leaves them abstract; they could be replaced by particular operations such as DES and RSA [29].
- `A` has its own name and `B`'s name, and `S` has its own name — here "name" can be synonymous with "network address".
- `A` and `B` have `S`'s public key `KpS`, `A` has the newly created symmetric key `KsA`, `B` has the newly created symmetric key `KsB`, and `S` has its own private key `^KpS`.
- `A` and `B` can be confident that `KpS` is `S`'s public key, `S` can be confident that `^PkS` is `S`'s private key, `A` can be confident that `KsA` was created for the current run of the protocol and will make a suitable shared secret for `A` and `S`, and `B` can be confident that `KsB` was created for the current run of the protocol and will make a suitable shared secret for `A` and `B`.
- `A` and `B` trust each other and `S`, and `S` trusts `B`.

In the specification of the protocol itself, `{x}f(y)` denotes the encryption of `x` with the function `f` and the key `y`, and `||` binds a message to a statement to indicate that the protocol assumes the message will not be sent unless the sender has adequate reason to believe this statement.

The goals for the protocol are the following:

- After stage 1, `S` has received `KsA`.
- After stage 3, `S` has received `KsB` and believes that `KsB` is a suitable shared secret for `A` and `B`.
- After stage 4, `A` has received `KsB` and believes that `KsB` is a suitable shared secret for `A` and `B`.

ISL allows C-style comments; the AAPA ignores them.

```
/* Tatebayashi-Matsuzakai-Newman Protocol */

DEFINITIONS:
PRINCIPALS: A,B,S;
PRIVATE KEYS: ^KpS;
PUBLIC KEYS: KpS;
SYMMETRIC KEYS: KsA,KsB;
ENCRYPT FUNCTIONS: Se,Sd,Pe,Pd;
Se WITH ANYKEY HASINVERSE Sd WITH ANYKEY;
Pe WITH KpS HASINVERSE Pd WITH ^KpS;

INITIALCONDITIONS:
A Received Se,Sd,Pe,Pd,A,B,KpS,KsA;
```

```
A Believes
  (Fresh KsA;
   PublicKey S Pe KpS;
   SharedSecret A S KsA;
   Trustworthy B;
   Trustworthy S);
B Received Se,Sd,Pe,Pd,KpS,KsB;
B Believes
  (Fresh KsB;
   PublicKey S Pe KpS;
   SharedSecret A B KsB;
   Trustworthy A;
   Trustworthy S);
S Received Se,Sd,Pe,Pd,S,^KpS;
S Believes
  (PrivateKey S Pd ^KpS;
   Trustworthy B);

PROTOCOL:
1. A->S: {KsA}Pe(KpS),A,B;
2. S->B: S,A;
3. B->S: {KsB}Pe(KpS)
          ||(SharedSecret A B KsB);
4. S->A: {KsB}Se(KsA)
          ||(SharedSecret A B KsB);

GOALS:
1. S Possesses KsA;
3. S Possesses KsB;
   S Believes SharedSecret A B KsB;
4. A Possesses KsB;
   A Believes SharedSecret A B KsB;
```

4 Definition of ISL

This section gives the full definition of ISL. It gives the BNF grammar for ISL, with interspersed comments. These comments include notes on semantic restrictions imposed by the AAPA, error messages produced by the AAPA, and informal descriptions of the meanings of ISL constructs asserting properties of principals and data items.

A protocol specification has four parts: definitions giving the principals, functions, and data items in the protocol; the protocol's assumed initial conditions; the protocol itself; and finally the authentication conditions the protocol is expected to satisfy.

```
<ProtSpec> ::= <Defs> <Init> <Prot> <Goals>
```

The definitions section lists the protocol's principals, the keys they use, other data items such as timestamps and nonces that they use, and definitions of the functions they apply and relationships among these functions.

```
<Defs> ::=
  'DEFINITIONS:'
  <Principals>
  [<KeySetSpecLst>]
  [<OtherTrm>]
  <FunSetSpecLst>
  [<RelSpecLst>]
```

Principals are given by identifiers. ISL identifiers are standard in that they can contain numeric characters if these numeric characters are preceded by alphabetic characters, but nonstandard in that they can begin with one of the special characters ~ or ^. These special characters are intended to mean "not", so they can be used to construct intuitive names for the private key corresponding to a public key or the inverse of an invertible function. In its output files, the AAPA translates both ~ and ^ to UN.

```
<Principals> ::= 'PRINCIPALS:' <IdLst> ';'

<IdLst> ::= <Id> [',' <IdLst>]

<Id> ::= ['~' | '^'] <Chr> [<ChrOrDgtLst>]

<Chr> ::= 'A'|...|'Z'|'a'|...|'z'

<ChrOrDgtLst> ::=
 ( <Chr> | <Dgt>) [<ChrOrDgtLst>]

<Dgt> ::= '0'|...|'9'
```

The keys in the protocol must be identified as private, public, or symmetric. This information guides the AAPA's proof process; see [3].

```
<KeySetSpecLst> ::=
 <KeySetSpec> [<KeySetSpecLst>]

<KeySetSpec> ::=
 <KeyType> 'KEYS:' <IdLst> ';'

<KeyType> ::=
 'PRIVATE' | 'PUBLIC' | 'SYMMETRIC'
```

Non-key items such as nonces and timestamps are lumped together.

```
<OtherTrm> ::= 'OTHER:' <IdLst> ';'
```

The functions in the protocol must be identified as encryption, hash, key-exchange, or "other" functions. "Encryption" includes both encryption and decryption.

"Other" functions include things such as adding or subtracting 1 to change a piece of data in a reversible way. Hash and "other" functions can take keys as additional arguments; a hash function taking a key as an additional argument computes message authentication codes.

```
<FunSetSpecLst> ::=
 <FunSetSpec> ';' [<FunSetSpecLst>]

<FunSetSpec> ::=
 <FunType> 'FUNCTIONS:' <IdLst>

<FunType> ::=
   'ENCRYPT'
 | ['KEYED'] 'HASH'
 | 'KEYEXCHANGE'
 | ['KEYED'] 'OTHER'
```

The "relations specification list" gives the inverses of invertible functions and the identities expressing the critical properties of key-exchange functions. A "relation specification" relates a function, function with key, or function applied to a list containing private and public keys, to another function, function with key, or function applied to a list containing private and public keys.

If the relation specification says that one function is the inverse of another, both functions must be given with or both without keys; if both are given with keys, the `ANYKEY` value can be used as the key for both functions. For public-key encryption, the keys, and possibly the functions, will be different. For symmetric-key encryption, the keys (or `ANYKEY` values) will be the same, though the functions might be different. Representative examples in the various cases are

```
Minus1 HASINVERSE Plus1;
DES WITH ANYKEY HASINVERSE DES WITH ANYKEY;
RSA WITH KpA HASINVERSE RSA WITH ^KpA;
```

If the relation specification says that one function value is equal to another, the same function must be given on both sides of the `EQUALS`, the function must have been declared as a key-exchange function, and this function must be given with lists of arguments whose first elements are private keys and whose second elements are public keys.

```
<RelSpecLst> ::= <RelSpec> ';' [<RelSpecLst>]

RelSpec ::=
  <Id>
  ['WITH' <GenKey>]
  <RelType>
  <Id>
  ['WITH' <GenKey>]

<GenKey> ::=
 <Id> | '(' <IdLst> ')' | 'ANYKEY'

<RelType> ::= 'HASINVERSE' | 'EQUALS'
```

The initial-conditions section is syntactically an arbitrary list of statements, though the proof attempt will fail with an error message if this list of statements includes other than `Believes` or `Received` statements.

```
<Init> ::= 'INITIALCONDITIONS:' <StmtLst>

<StmtLst> ::= <Stmt> ';' [<StmtLst>]
```

Informal descriptions of the meanings of the various statement constructors follow. These constructors' formal meanings are given via the function `BGNY` defined in the AAPA's underlying HOL theory `bgny`; `BGNY` assigns truth values to `:Statement` objects. Theory `bgny` differs in only relatively minor, technical ways from the theory `protocol` described in [4]; [3] describes these differences.

- `Believes`: A principal has adequate reason to believe a statement.
- `Conveyed`: A principal was the creator and source of a data item.
- `Fresh`: A data item was created for the current run of the protocol.
- `NeverMalFromSelf`: Either two principals are equal, in which case a data item is supposed to be a replay of something the first principal sent out earlier, or the two principals are not equal and the first principal has adequate reason to believe this data item was not part of any message that the first principal sent out at any past time. (The name means, "never maliciously from self".)
- `Possesses`: A principal received a data item or is capable of computing it from items that this principal has received.
- `PrivateKey`: A key, for an algorithm, is one of a principal's private keys.
- `PublicKey`: A key, for an algorithm, is one of a principal's public keys.
- `Received`: A principal received a data item before the current session, or received this data item as, or as part of, some message sent earlier in the current session.

- `Recognizes`: A principle can identify a data item as being meaningful information of an expected form.
- `SharedSecret`: If two principals possess a data item, or come to possess it through secure means, then they are or will be the only ones other than principals trusted by both of them who possess it.
- `Trustworthy`: If a principal was the source of a data item with an associated statement, then this is adequate reason for believing that the associated statement is true.

Statements are defined in terms of "statement sublists", which are lists of implicitly conjuncted statements separated by semicolons, and by lists of terms, which correspond to lists of data items.

```
<Stmt> ::=
   <Id> 'Believes' <Stmt>
 | <Id> 'Conveyed' <TrmLst>
 | <Id> 'Creates' <TrmLst>
 | 'Fresh' <TrmLst>
 | 'NeverMalFromSelf' <Id> <TrmLst>
 | <Id> 'Possesses' <TrmLst>
 | 'PrivateKey' <Id> <Id> <Id>
 | 'PublicKey' <Id> <Id> <Id>
 | <Id> 'Received' <TrmLst>
 | <Id> 'Recognizes' <TrmLst>
 | 'SharedSecret' <Id> <Id> <TrmLst>
 | 'Trustworthy' <Id>
 | '(' <StmtSbLst> ')'

<StmtSbLst> ::= <Stmt> [';' <StmtSbLst>]

<TrmLst> ::= <Trm> [',' <TrmLst>]
```

A term is either an identifier — a principal name, key, nonce, password, timestamp, etc. — or an identifier previously declared as a function applied to a list of arguments, or an expression denoting signed or encrypted data. The term `<x>h(k)` denotes `x` together with a message authentication code produced from `x` with key-dependent hash function `h` and key `k`. The term `[x](h,f)(k)` denotes `x` together with a signature that is the encryption, using function `f` and key `k`, of the hash of `x` produced using non-key-dependent function `h`. Finally, `{x}f(k)` denotes `x` encrypted with function `f` and key `k`.

Every hashed, encrypted, or signed term can optionally have an associated *extension*, which is a statement the protocol expects a principal to believe or else it would not release this term. For signed terms, the AAPA associates the extension for the term with the message authentication code or encrypted hash code that serves as the signature part of the term. The term `t||(slist)` denotes term `t` with the implicitly conjuncted list of statements `slist` as an extension. The ISL treatment of extensions comes from the GNY logic [10], but it restricts extensions to hashed or encrypted data, since only hashed or encrypted data is protected from being secretly modified in transit, only data that cannot be secretly modified in transit can be confidently identified as coming from a particular principal, and only data coming from a particular principal can be confidently believed to have the properties the protocol expects it to have.

Each encrypted or hashed term used in specifying initial conditions must have a `From` construct identifying a putative source principal for this term. With this information, the AAPA is able to assign a putative source to every piece of information in the protocol whose true source could be confidently identified. The proof process does not assume that these putative sources are correct, but it uses them to direct which conveyance theorems it tries to prove.

```
<Trm> ::=
  <Id>
 | <Id> '(' <TrmLst> ')' [<Ext>]
 | '<' <TrmLst> '>'
   <Id>
   '(' <TrmLst> ')' [<Ext>]
 | '[' <TrmLst> ']'
   '(' <Id> ',' <Id> ')'
   '(' <TrmLst> ')' [<Ext>]
 | '{' <TrmLst> '}'
   <Id>
   '(' <TrmLst> ')' [<Ext>]
 | <Trm> 'From' <Id>

<Ext> ::= '||' ['(' <StmtSbLst> ')']
```

The protocol proper is a list of messages, separated by semicolons. Each message is given by a header giving the protocol stage, the sending principal, the receiving principal, and the data sent.

```
<Prot> ::= 'PROTOCOL:' <MsgLst>

<MsgLst> ::= <Msg> ';' [<MsgLst>]

<Msg> ::= <Hdr> ':' <TrmLst>

<Hdr> ::= <Nmbr> '.' <Id> '->' <Id>

<Nmbr> ::= '1'|...|'9'  [<DgtLst>]

<DgtLst> ::= <Dgt> [<DgtLst>]
```

Finally, the goals section gives the protocol's expected properties, by protocol stage, as statements. A stage can have no associated statements, in which case it can be omitted, or it can have several associated statements, in which case these statements are given in a list, with each statement terminated by a semicolon. While stages can be omitted, they must be given in increasing order.

```
<Goals> ::= 'GOALS:' <GoalStmtLst>

<GoalStmtLst> ::= <GoalStmt> [<GoalStmtLst>]

<GoalStmt> ::= <Nmbr> '.' <StmtLst>
```

5 AAPA Outputs

This section gives a brief description of the outputs produced by the AAPA. See [2] for detailed descriptions of using the AAPA to iteratively examine and modify the ISL specifications of three complicated protocols.

When run on a file containing a syntactically and semantically correct ISL specification, the AAPA tries to prove, by induction on protocol stage, that the protocol has all its specified properties. The AAPA produces terminal output describing its progress in constructing proofs, and, typically, ISL output files describing the results that it could and could not prove. If it proves that a protocol has all its specified properties, the AAPA outputs a terminal message saying that it did not find any problems with the protool; it outputs ISL files in this case only if the user requested them with an optional command-line argument. If it cannot prove that a protocol has all its specified properties, the AAPA outputs a terminal message identifying the first such proof failure, and produces ISL files describing the results that it could and could not prove.

The AAPA proves two types of goals: *default* goals, which are mechanically generated goals typically having true specified protocol properties as trivial consequences, and *user* goals, which are goals derived from the `GOALS:` section of the ISL specification. When it fails to prove a user goal, the AAPA produces a `.fail` file describing all the default goals that it could not prove, and a `.prvd` file describing all the theorems that it did prove, before encountering the failure.

If the ISL specification of the Tatebayashi, Matsuzakai, Newman protocol from Section 4 is placed in a file named `tmn.isl`, for example, running the AAPA on `tmn.isl` produces the terminal output:

```
Creating theory tmn
Beginning tmn proofs
Initializing globals
Proving default goals, stage 1
Retrying failed default goals, stage 1
Proving user goals, stage 1
Proving default goals, stage 2
Proving user goals, stage 2
Proving default goals, stage 3
Retrying failed default goals, stage 3
Proving user goals, stage 3

User-goal failure, stage: 3!

Goal statement:
 S Believes (SharedSecret A B KsB);
```

The AAPA produces files `tmn.fail` and `tmn.prvd`. File `tmn.fail` immediately shows the problem:

```
/* ## Failed default goal from stage 3: ## */

S Believes
 (B Conveyed
  {KsB}Pe(KpS)||(SharedSecret A B KsB));

/* ========== Waiting subgoals: ========== */

S Believes (S Recognizes KsB);
S Believes (PrivateKey S Pd UNKpS);
S Believes (SharedSecret S B KsB);
S Believes
 (Fresh
  {KsB}Pe(KpS)||(SharedSecret A B KsB));

/* =========== Proved subgoals: ========== */

S Received
 {KsB}Pe(KpS)||(SharedSecret A B KsB);
S Possesses Pd,UNKpS;
```

Stage 3 of the protocol sends `KsB` to `S`, and protects it against disclosure by encrypting it with `S`'s public key. The protocol assumes that `B` will not send the message it sends in stage 3 unless `B` believes that `KsB` is a suitable shared secret for `A` and `B`, and `B` does believe this. Further, `S` trusts `B`, so `S` can believe that `KsB` has this property if `S` can believe that the message it receives in Stage 3 is from `B`.

But `S` has no reason to believe this message comes from `B`. Since `S` cannot identify `KsB` as meaningful information — keys are carefully chosen to be as random as possible — even a principal *not having* `S` *'s public key* could send `S` random information that `S` could decrypt to obtain a seemingly acceptable key.

If this problem were eliminated, say by changing the protocol to put recognizable information, such as a name, in with `KsB` before encrypting it with `S`'s public key, the AAPA would automatically prove the new version of the first waiting subgoal and then prove the second waiting subgoal.

The third waiting subgoal would still be a problem, though. `S`'s public key is widely known, so principals other than `B` could have sent `S` a key, unless `S` and `B` happened to share `KsB` as a secret in the first place. This is obviously not the intent of the protocol.

The fourth waiting subgoal would also still be a problem. Even if the message contained something that was a secret shared between `S` and `B`, that something could be a replay of something `S` itself sent out, possibly acting in a different role, in an earlier run of the protocol. There must be something in the encrypted data that `S` can identify as created for the current run of the protocol before `S` has adequate reason to believe that the message comes from `B`.

The requirement that `S` be able to identify something in the message as having been created for the current run of the protocol would also come up in having `S` believe that `KsB` has its expected properties, even if `S` could otherwise believe that the message sent to it in stage 3 was from `B`. The statements associated with messages in a protocol only hold for the *current* session. Otherwise, the whole message `S` receives in stage 3 could be a replay of a message `B` sent `S`, for the purpose of communicating with `A`, months ago, during which time the attacker has been able to compute `KsB` by examining messages sent between `A` and `B`.

The outputs of the AAPA analysis thus show that, despite its apparent simplicity, the Tatebayashi, Matsuzakai, Newman protocol actually has several deep vulnerabilities, some of which cannot be eliminated without changing the basic nature of the protocol.

The only skill required in using the AAPA, beyond that of writing ISL specifications that correctly capture protocol designs and their expected properties, is interpreting the AAPA's outputs to figure out what went wrong when problems arise. This process is very similar to debugging. It requires a basic understanding of the AAPA's proof process and the rules of its belief logic. This understanding can be obtained by studying material on the AAPA [4, 3], studying sample AAPA analyses [2], and gaining experience with using the AAPA's `.fail` and `.prvd` files.

6 Relations to Other Tools

This section describes the AAPA's strengths and weaknesses in relation to other protocol analysis tools.

Current approaches to avoiding protocol failure either attempt to construct possible attacks, using algebraic properties of the algorithms in the protocols, or attempt to construct proofs, using specialized logics based on a notion of "belief", that protocol participants can confidently reach desired conclusions.

The main attack-construction tools are Millen's Interrogator [20, 22, 21], Meadows' NRL Protocol Analyzer [17, 18, 19], and the CSP model-checking tool FDR [26] used with techniques developed by Roscoe and Lowe [27, 28, 15].

In principal, the attack-construction tools do a more thorough job than the AAPA does of identifying possible protocol failures. They address nondisclosure as well as authentication, and can model attacks based on exploiting algebraic properties of the encryption and decryption operations used, or on legitimate protocol participants' only checking some fields of some messages for some properties. See [32, 15, 28] for examples and discussions of such attacks.

The popular RSA public-key encryption/decryption algorithm [29], for instance, has the property that, roughly speaking, the encryption of a product is the product of the encryptions. If RSA is used for both the encryption and decryption operations `Pe` and `Pd` for the Tatebayashi, Matsuzakai, Newman protocol described in Section 3, this gives the following vulnerability that the AAPA does not detect:

The attacker can eavesdrop on stage 3 of the protocol, obtaining `{KsB}RSA(KpS)`, then compute a new random symmetric key `KsX` and use the algebraic property of RCS, with `S`'s known public key `KpS`, to compute `{KsB*KsX}RSA(KpS)`. The attacker can then run the protocol, playing the roles of both `A` and `B` and using `KsB*KsX` in place of `KsB`, to obtain `KsB*KsX` from `S`. This allows the attacker to compute `KsB` and decrypt all the subsequent communication between `A` and `B`.

For practical purposes, though, particularly in designing long or complicated protocols, the AAPA has advantages over the attack-construction tools. They search attack spaces that grow exponentially with the size of the protocol, requiring extensive, expert, sometimes prolonged user guidance. Even with this guidance, they can require unavailable amounts of time or space. In contrast, the AAPA does no searching. The time it requires to analyze a protocol grows quadratically, not exponentially, with the size of the protocol. It requires no user guidance. Under realistic conditions, it never requires unavailable amounts of time or space, and it produces its results in minutes.

If an attack-construction tool's search is interrupted, the only conclusion is that "there might be a problem with the protocol, but the analysis did not find it". This is similar to the conclusion when the AAPA proves all the user-set goals. The AAPA's analysis in this case, though, shows that there are none of the types of failures that the AAPA detects — e.g., failures arising from replay attacks, inadequate trust conditions, and inadequate freshness or recognizability properties — for the full protocol. The attack-construction tool's interrupted search does not eliminate any possible attacks, even elementary ones, for the full protocol.

If an attack-construction tool's search terminates without finding an attack, a bug in the tool could also have prevented it from finding an attack that it should have found. Because the AAPA's proofs are carried out with HOL, and because being a theorem in HOL is determined by *type checking* in a Standard ML compiler [25], the accuracy of the AAPA's theorems is *not* dependent on the accuracy of the AAPA's proof process. Only errors in its belief logic can cause the AAPA to miss failures that it should have found. This point is discussed more fully in [4].

The attack-construction tools also vary in their strengths and weaknesses:

- When the NRL Protocol Analyzer generates and searches an attack space for a protocol without finding an attack, this constitutes a proof that no attack exists within a very broad class of attacks. It is constrained to protocols using only operations defined by *confluent* rewrite systems — i.e., systems having the property that for any term, if all the possible rewrites are applied to the term in any order, the process terminates and the result does not depend on the order in which the rewrites were applied. This prevents it from being used to analyze protocols using symmetric operations such as exclusive-or or Diffie-Hellman key exchange. It also uses an inefficient search process.

- The Interrogator can analyze protocols using operations defined by non-confluent rewrite systems, but its failure to find an attack does not constitute a proof that no attack exists within its model. It also uses an inefficient search process.

- FDR can analyze protocols using operations defined by non-confluent rewrite systems, and it uses pre-computation of a large set of potentially useful values available to an attacker to give it a much more efficient search process. It requires that the user set limits in advance on things such as the maximum number of objects of various types that it will consider, though, so its failure to find an attack shows only that an attack does not exist within the restrictions set by the user.

This paper will not list the non-AAPA proof construction tools, just identify two of them and tell where to find references to others. Mathuria, Safavi-Naini, and Nickolas [16] developed a tool that, like the AAPA, automatically constructs proofs from a variant of the GNY logic; their paper contains references to provers based on the earlier, simpler, less expressive belief logic developed by Burrows, Abadi, and Needham [6]. Schubert [30] developed a partial implementation, with some automated proof support, of the Syverson and van Oorschot belief logic [33]. None of these tools use a simple specification language such as ISL, and none have an efficient, fully automatic proof-construction procedure.

Millen is developing the Common Authentication Protocol Specification Language (CAPSL), partly inspired by ISL, for use with *all* the protocol analysis tools. This work has not yet been published in a form available for indefinite future reference, but as of February, 1997 it is described in the World Wide Web site at `http://www.mitre.org/research/capsl`.

7 Future Extensions

This section lists several weaknesses of the AAPA, and work in progress to lessen or eliminate them. It ends with a description of a possible proof-based protocol analysis tool capable of making protocol failure into essentially a solved problem.

This section considers the following AAPA weaknesses:

1. The AAPA does not detect nondisclosure failures.

2. The AAPA does not detect failures involving algebraic properties of algorithms.

3. The AAPA does not detect failures involving multiple executions of protocols, or attacks that trick legitimate protocol participants into performing computations for the attacker.

4. The AAPA does not construct actual attacks exploiting the vulnerabilities it finds.

5. The AAPA does not prove all the results potentially provable from its belief logic.

Weaknesses 1, 2, and 3 involve the AAPA's belief logic, Weakness 4 involves the AAPA's use with other protocol analysis tools, and Weakness 5 involves the AAPA's proof-construction procedure.

Weakness 1 can be addressed by replacing constructor `SharedSecret` with a constructor `KnownOnlyTo`, and defining appropriate inference rules and proof procedures for this new constructor.

Weakness 2 can be addressed by generalizing the ISL `EQUALS` construct to express arbitrary equalities.

Weakness 3 can be addressed, in large part, by replacing the `NeverMalFromSelf` construct with a *computed* test for whether a principal might be tricked into thinking a message received at a certain stage, expected to be of a certain form, might actually have been sent out at a different stage, in a different execution of the protocol, in a different form, by the principal itself. This would require additional computation, but not an exponential search.

Weakness 4 can be addressed by using the AAPA as an efficiency-improving front end to an attack-construction protocol analysis tool, using the AAPA's analysis to direct the attack-construction tool's search. An effort is under way to do this for the NRL Protocol Analysis Tool. Weakness 4 is seldom critical in practice, since a human analyst can typically construct an attack from the information in the AAPA's `.fail` and `.prvd` files.

Weakness 5 can be addressed by complicating the AAPA's proof construction process. This weakness is actually of low priority, though, since the AAPA already proves all naturally stated provable user goals; see [3] for a discussion of this issue.

Snekkenes [32] has suggested a third alternative to protocol analysis based on attack construction or proof using a belief logic: formally defining a model of protocols and attacks on them that is as rich as any model used by an attack-construction tool, and then proving, using a general-purpose theorem prover such as HOL, that protocols are secure over this model. Meadows has suggested that the NRL Protocol Analyzer's search-space-defining algorithm might be exactly what is needed to automate the construction of such proofs.

If Meadows is right, and the proof process for a protocol does not contain what is effectively a search over a space exponential in the size of the protocol, then all the rigor and completeness of attack construction can be produced with speed comparable to that of belief-logic analyses. Perhaps the rules of inference for the belief logics correspond to useful, provable lemmas for this more inclusive proof construction process. Work is currently in progress to answer these questions.

8 Summary

The following statements summarize this paper:

- Protocol failure is a significant problem that affects the designers of any network service requiring secure communication.
- The AAPA is a tool that takes easy-to-write ISL specifications of protocols and their expected properties, and very quickly performs formal analyses checking for failures of these protocols to achieve their desired properties.
- The AAPA does not detect all possible protocol failures, but detects a large family of common errors, and detects them quickly and easily enough to make getting an AAPA analysis a valuable part of any protocol-creation or protocol-selection process.
- The AAPA has strengths and weaknesses that are comparable to the strengths and weaknesses of the best current protocol analysis tools; it is currently the best suited of all these tools for aiding in the design process.
- Work is under way to significantly reduce the AAPA's weaknesses without significantly reducing its strengths.
- Future proof-based analysis tools could effectively solve the protocol-failure problem.

References

[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 122–136, Oakland, CA, 1994. IEEE.

[2] S. Brackin. Automatic formal analyses of cryptographic protocols. In *Proceedings of the 19th National Conference on Information Systems Security*, Baltimore, MD, October 1996. IEEE.

[3] S. Brackin. Deciding cryptographic protocol adequacy with HOL: The implementation. In *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 61–76, Turku, Finland, August 1996. Springer-Verlag.

[4] S. Brackin. A HOL extension of GNY for automatically analyzing cryptographic protocols. In *Proceedings of Computer Security Foundations Workshop IX*, County Kerry, Ireland, June 1996. IEEE.

[5] S. Brackin and R. Lichota. CASE for high assurance: Utilizing commercial technology for automatic cryptographic protocol analysis. In *Dual-Use Technologies and Applications Conference, Information Management Collections Processing and Distribution*, Utica, NY, June 1996. IEEE.

[6] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, February 1990.

[7] B. Cox, J. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *Proceedings of the First Workshop on Electronic Commerce*, 1995.

[8] D. Denning and G. Sacco. Timestamps in key distribution protocols. *CACM*, 24(8):533–536, August 1981.

[9] L. Gong. Handling infeasible specifications of cryptographic protocols. In *Proceedings of Computer Security Foundations Workshop IV*, pages 99–102, Franconia NH, June 1991. IEEE.

[10] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 234–248, Oakland, CA, May 1990. IEEE.

[11] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, 1993.

[12] R. Lichota, S. Brackin, and G. Hammonds. Verifying the correctness of authentication protocols using "Convince". In *Proceedings of the 12th Annual Computer Security Applications Conference*, San Diego, CA, December 1996. ACM.

[13] R. Lichota, G. Hammonds, and S. Brackin. Verifying cryptographic protocols for electronic commerce. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, Oakland, CA, November 1996. IEEE.

[14] G. Lowe. Comments in panel discussion "What is an attack on a cryptographic protocol?". Computer Security Foundations Workshop IX, June 1996.

[15] G. Lowe. Some new attacks upon security protocols. In *Proceedings of Computer Security Foundations Workshop IX*, County Kerry, Ireland, June 1996. IEEE.

[16] A. Mathuria, R. Safavi-Naini, and P. Nickolas. Some remarks on the logic of Gong, Needham and Yahalom. In *Proceedings of the International Computer Symposium*, pages 303–308, Hainchu, Taiwan, December 1994. NCTU.

[17] C. Meadows. Using narrowing in the analysis of key management protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 138–147, Oakland, CA, May 1989. IEEE.

[18] C. Meadows. A system for the specification and analysis of key management protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 182–195, Oakland, CA, May 1991. IEEE.

[19] C. Meadows. Applying formal methods to the analysis of a key management protocol. *J. Computer Security*, 1(1):5–36, 1992.

[20] J. Millen. The interrogator: A tool for cryptographic protocol analysis. In *Proceedings of the Symposium on Security and Privacy*, pages 134–141, Oakland, CA, May 1984. IEEE.

[21] J. Millen. The Interrogator model. In *Proceedings of the Symposium on Security and Privacy*, pages 251–260, Oakland, CA, May 1995. IEEE.

[22] J. Millen, S. Clark, and S. Freedman. The Interrogator: Protocol security analysis. *IEEE Trans. on Software Engineering*, SE-13(2):274–288, February 1987.

[23] J. Moore. Protocol failure in crypto systems. *Proceedings of the IEEE*, 76(5):594–602, May 1988.

[24] C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[25] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1993.

[26] W. Roscoe. Model-checking CSP. In W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare.* Prentice-Hall, 1994.

[27] W. Roscoe. Modeling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of Computer Security Foundations Workshop IX*, County Kerry, Ireland, June 1995. IEEE.

[28] W. Roscoe. Intensional specifications of security protocols. In *Proceedings of Computer Security Foundations Workshop IX*, County Kerry, Ireland, June 1996. IEEE.

[29] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, New York, 1995.

[30] T. Schubert and S. Mocas. A mechanized logic for secure key escrow protocol verification. In *Higher Order Logic Theorem Proving and Its Applications*, number 971 in Lecture Notes in Computer Science, pages 308–323, Aspen Grove, UT, September 1995. Springer-Verlag.

[31] M. Sirbu and J. Chuang. Distributed authentication in Kerberos using public key cryptography. In *Internet Society Symposium on Network and Distributed System Security*, San Diego, CA, February 1997. IEEE.

[32] E. Snekkenes. *Formal Specification and Analysis of Cryptographic Protocols*. PhD thesis, University of Oslo, Oslo, Norway, January 1995.

[33] P. Syverson and P. van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of the Symposium on Security and Privacy*, pages 14–28, Oakland, CA, 1994. IEEE.

[34] N. Tatebayashi, N. Matsuzakai, and D. Newman. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology — CRYPTO '89*, number 435 in Lecture Notes in Computer Science, pages 324–333, New York, 1990. Springer-Verlag.